Basics of Programming in OCAML

In this section we will learn some of the basic features of programming in OCAML, or functional programming languages like it in general. Here is a summary of the things we will look at:

- Local variables/bindings
- Functions
- Tuples
- Lists
- Functions as values and higher-order functions
- Recursion on functions
- Pattern matching with lists
- The option type
- Custom types

Local variables

In every programming language we need some way to retain information for later use, by associating some symbol/string/identifier with it. A key part of that assignment is a specification of the rules for the *scope* of that assignment, namely the extent of the code where it would apply.

In OCAML we do this using `let`:

```
let x = 3 + 5 in x + x;;
```

This introduces a new binding of `x` to the result of the computation of `3 + 5`. The scope of the binding is the expression that follows the `in`. You can even nest bindings:

```
let x = 3 + 5
in let y = x + 2
   in x + y;;
x;;    (* x is undefined here! *)
```

When you work at the interactive level of `utop`, there is one further thing you can do, which is to define something that persists across the duration of your interactive session:

```
let x = 3 + 5;;
(* x equals 8 from now on *)
```

Functions

Functions are extremely versatile in OCAML.

Function calls

To call a function you simply put its argument next to it, no parentheses necessary:

```
sin 2.0;;
```

Chaining function calls does need parentheses, but only to signify what needs to happen first:

```
cos sin 2.0;;     (* error! *)
cos (sin 2.0);;   (* right! *)
```

Some functions appear to take 2 arguments. Technically they do not, but we will worry about that later. multiple arguments are simply listed next to each other:

```
max 2 4;;
```

Perhaps now you can guess what happened with the error earlier when we tried to chain functions.

Function definitions

Function definitions work exactly like any other values. In fact as far as OCAML is concerned there isn't really a difference between defining a function and defining a constant, other than a little bit of syntactic sugar. Here's how we could define a simple function that turns negative numbers to 0:

```
let xplus x = if x < 0 then 0 else x;;
xplus 2;;
xplus (-2);;
```

We also see here a use of OCAML's conditional, if–then–else. It takes exactly 3 expressions, one that must return a boolean, and the two branches that must return the same type.

Functions of multiple arguments are defined similarly:

```
let myMax x y = if x < y then y else x;;
```

OCAML also has anonymous functions. These are often called "lambdas":

```
let x -> x * x;;    (* A new function, but it gets "lost" because we didn't store it *)
(fun x -> x * x) 3;;    (* Here we immediately invoke the function! *)
(fun x y -> x * y) 3 4;;    (* Anonymous function of 2 variables! *)
let xplus = fun x -> if x < 0 then 0 else x;;    (* Alternative definition to above *)
```

Tuples

A powerful feature of OCAML is the ease with which you can group heterogeneous values together. This feature is called a tuple. Tuples can be used anywhere where other values can:

```
let minmax x y = if x < y then (x, y) else (y, x);;
minmax 4 3;;
```

Tuples can also be used as arguments to functions, which appear deceptively similar to functions in other languages:

```
let myAdd (x, y) = x + y;;
myAdd (2, 3);;
```

It might look like you have defined a function of two variables, but in fact you have not. It is a function of one variable, which happens to be a tuple, and we call its two components x and y. For instance we could get the exact same effect as:

```
let myAdd tpl = fst tpl + snd tpl;;
```

Here fst and snd are functions provided by OCAML to pick the first, respectively second, part out of a tuple.

Tuples can of course have more than 2 values, and they can have values of different types:

```
("a string", 5, 3.4);;
```

Lists

Lists are used for storing arbitrary numbers of values of the same type. You can think of them like linked lists, processed always in order.

```
[1; 4; 2];;  (* List of 3 elements *)
2 :: 4 :: [];;   (* elements appended to the empty list *)
```

There are two standard ways to go through list elements. One is pattern matching, and we will see more about that in a little while. The other is using the functions from the List module[1]:

```
let a = [1; 4; 2];;
List.length a;;
List.tl a;;
List.rev a;;
List.append a a;;
```

Most important amongst them are the so-called higher-order functions, called that way because they take functions as inputs, and then operate on the lists using those functions:

```
List.map (fun x -> x * x) a;;
List.filter (fun x -> x > 3) a;;
List.fold_left (fun acc x -> acc + x) 0 a;;
List.fold_left (+) 0 a;;          (* operators are really just normal functions *)
```

These are extremely versatile. The last one in particular, List.fold_left and its cousin List.fold_right, are sufficient to replace most accumulator-type problems that you would normally use a for loop for. In fact you can do most anything you could have done with a for loop via these folds.

Most "collection" types have these higher order functions implemented. They offer a uniform way to approach collection element traversal.

[1] http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html

Functions as values and Higher-order functions

Functions are ubiquitous in OCAML, in the sense that they behave in all ways like any other normal value. They can be passed around, stored in variables, be the input to other functions and so on.

The function `List.map` earlier is a simple example. It took as input a function that transformed elements of the list, and also a list, then applied that function to each element. This is a powerful "separation of concerns" paradigm: One function, `List.map` is responsible for knowing how to visit each element in the list, and another function, the passed-in argument, knows what to do with those elements. The two don't really need to know much about each other.

Here is another example: We will define a function called "twice". It takes as input a function `f`, then returns as a result a function that would apply `f` twice. So `(twice f) 4` would be the same as `f (f 4)`. In fact this is essentially the definition:

```
let twice f = fun x -> f (f x);;
```

It is worth looking at the type of this function: `('a -> 'a) -> 'a -> 'a`. The `'a` stands here for any type, but it must be the same type throughout. It is called a polymorphic type. The arrows indicate that this is a function, and you can effectively think of it as a function of two arguments, in the line above those would be the `f` and the `x`. The `f` is required to be a function that takes `'a` things and returns `'a` things, and `x` is required to also be a `'a` thing. Finally our function returns an `'a` thing.

So this way of thinking would correspond to defining the functions thus:

```
let twice f x = f (f x);;
```

There is practically no difference between these two forms, the latter is really just "syntactic sugar" for the former, and actually that in turn is syntactic sugar for something like:

```
let twice = fun f -> fun x -> f (f x);;
```

All these 3 are the same. In general this is what we call a "curried" function, in honor of Haskell Curry. The idea is that a function of 2 arguments is intead thought of as a function of the first argument, that in turn returns a function of the second argument. We in effect apply the arguments one at a time.

Recursion on functions

We now arrive at an area where OCAML differs a lot from other languages. Not so much because other languages do not have recursive functions, but because of how ubiquitous they are. There are three main reasons for this:

- As a functional programming language, OCAML discourages mutation, i.e. the idea that you have a variable whose value you keep updating. This idea is required for iteration in most programming languages (iteration via `for` loops, `while` loops etc). So OCAML essentially does away with iteration altogether, and

uses recursive functions instead, who can serve a similar purpose as we will see shortly.

- OCAML, as most functional programming languages, has properly tail-optimized calls. What this means is that if a function calls another function as the last thing it does, i.e. its return value is just what that other function returns without need for extra steps, then these calls don't take up extra stack space on the function call stack. As an example, this silly function `let rec f x = f x;;` which just calls itself over and over again will just run forever rather than exhaust the stack space.

  This is useful to create things like interactive loops. Here is a "simple" example:

```
let rec loop () =
    let x = read_int ()
    in if x > 5
       then ()
       else loop ()
in loop ();;
```

  This loop reads an integer input, and if that is not bigger than 5 then it just asks for new input. This behaves very much like a `while` loop.

- In OCAML, again as in most functional programming languages, function values, which is effectively most of the information in the function's "activation record" that goes in the function stack, are actually stored in the heap, just like any other values. This removes the usual limitations that come from using a stack.

So how do we indicate recursive functions? Simple: just add the word `rec` before the function definition, like in the loop examples above.

Let us do another example. A standard problem in number theory is how to compute a large power of a number `a`, say the `m`th power, modulo an integer `n`. There is a trick that relies on doubling the value, and basically goes like this:

1. If `m` is even, then instead of `a^m` we compute `(a * a) ^ (m / 2)`. We compute the `a * a` modulo `n`, and also do a modulo `n` at the end.
2. If `m` is odd, then we do the same (where `m / 2` rounds down), but further multiply the result by `a`.
3. Finally if `m =1` we just return `a`.

So here is how this might look in OCAML.

```
let rec modpower a m n =
    if m = 1
    then a
    else let rest = modpower (a * a mod n) (m / 2) n
         in if m mod 2 = 1
            then a * rest mod n
            else rest;;
```

Pattern matching

Recursion works really well with another awesome feature of OCAML, pattern matching. Pattern matching is somewhat of a beefed-up version of a switch statement. Basically you match a value against a series of potential "patterns", until you find one that matches. These patterns can even contain variables, which then are bound to corresponding values for the duration of the call.

Let us see a simple example of that. Say we want to write a multiplication function that takes two numbers and multiplies them. But if one of the two numbers is 0 we want to return 0 directly without doing a multiplication. Don't ask me why, just go with it!. Perhaps we know that multiplication will for some reason take time, and we don't need it since we know the answer in this case. We further want to handle the case of 1 specially:

```
let myMult x y =
    match (x, y) with
      (0, _) -> 0
    | (_, 0) -> 0
    | (1, z) -> z
    | (z, 1) -> z
    | _      -> x * y
```

So let us see what happens here:

1. The pair (x, y) is formed, and matched against various "patterns".
2. First it is compared to the pattern (0, _). It will only match it if the x matches 0 and y matches _, which stands for "anything". So this will match if x is equal to 0, and in that case we return the value 0, which is next to the arrow. And we are done.
3. Otherwise, we look at the second pattern. That one says that x could be anything and y needs to equal 0. We return 0 in that event as well.
4. The third pattern is a bit more interesting. We could have used the same idea as in the first two, but I used this approach to make a point. So in that version, we have a variable name, z. What this means is that x would have to equal 1, and that y could be anything *but* for the expression on the right side the variable z will be bound to the value of y. So this effectively is the same as having written it as: (1, _) -> y.
5. The last clause is a "catchall", with a single underscore that will match anything. In that case we simply multiply x and y.

Okay, let us proceed to a more complicated example. Recursion together with pattern matching is an extremely powerful method for processing structured data, like a list. A list has one of two forms: It may be empty, or it may be an element prepended to another list. We can pattern match those two cases. For instance, here is how we could add the elements on a list:

```
let rec addAll lst =
    match lst with
        x :: rest -> x + addAll rest
      | []        -> 0
```

Here's an implementation of length, where for fun we have special-cased the list with one element. The order is important here: that case must come before the last one, as otherwise the last one would subsume it. Notice once again the usage of _ for values we don't really care for. It is a good practice to do that.

```
let rec length lst =
    match lst with
        []        -> 0
      | _ :: []   -> 1
      | _ :: rest -> 1 + length rest
```

Armed with this knowledge, let us see how we could define map and fold_left:

```
let rec map f lst =
    match lst with
        []        -> []
      | x :: rest -> (f x) :: map f rest;;

let rec fold_left f init lst =
    match lst with
        []        -> init
      | x :: rest -> fold_left f (f init x) rest;;
```

The option type

A cool feature of OCAML is the *option type*. The option type is a way, at the type system level, to handle the idea of failure. A value of type for instance int option has two alternatives:

1. It could correspond to an int value. In that case it is written like so: Some 5
2. It could correspond to "failure". In that case it is written like so: None

As a quick example, imagine the following "safeDiv" function for safe division:

```
let safeDiv x y =
    if y = 0
    then None
    else Some (x / y);;
```

So in this case if the "denominator" y is equal to zero, we do not attempt to divide. This way we can be sure that we will not have to deal with a runtime exception. The other advantage is that if someone wants to use our value, they have to explicitly do a pattern match and say "if I am in the case of None I do this, otherwise if I am Some n I can do something with the n". So anyone who wishes to use our value is forced by the type system to deal with the possibility that our function might not have returned an actual value. They can't simply forget to account for it. This is powerful.

Unfortunately by default there is no reach set of functions to work with the option type. In this section we will look at their implementation. At the same time we will learn a bit about OCAML's module system.

This time we will work with files. I have already created the two files for us, you will have to download them. I suggest that you check out the entire GitHub project[2]. Choose a directory you want to use, then do:

```
git clone https://github.com/skiadas/TheoryCompCourse
```

It will create a folder called TheoryCompCourse at that location. Later on, if you want to receive the updated versions, you simply go in that directory and do:

```
git pull
```

For now, go into the ocaml folder within that TheoryCompCourse folder, and you will find two files, option.ml and option.mli. Open them both in your favorite text editor.

Let us start by looking at options.mli. This is what is called an *interface* file. It is a specification of the functions that are in the module, and their types. You can actually create a lot more functions in the module, but they will be kept private and hidden to the rest of your program if they don't appear in the interface file.

So all that the interface file contains is type signatures for your functions. It can also contain definitions of new types, and definitions of exceptions. In fact in this case you will see an exception defined.

Here's a condensed version of the function signatures in the interface file:

```
val may : ('a -> unit) -> 'a option -> unit
val map : ('a -> 'b) -> 'a option -> 'b option
val default : 'a -> 'a option -> 'a
val map_default : ('a -> 'b) -> 'b -> 'a option -> 'b
val is_none : 'a option -> bool
val is_some : 'a option -> bool
val get : 'a option -> 'a
```

Oftentimes function signatures pretty much tell you what the function does. For instance, the first function, called may. It takes as first argument a function that expects some value of type 'a and returns nothing (unit is the type with only one value, (), so it is used to indicate that the function doesn't really return anything, it simply has a side-effect of say writing something on the screen). It then also takes as an argument a value of type 'a option. And finally it returns nothing. So it should not be hard to guess what the function does. If the argument of type 'a option has the form Some aVal, then we can call the function on aVal. If instead it is a None then we cannot do much. Here's how we would implement this function (omitting the semicolons because they are not used when working with files) see the option.ml file:

```
let may f v_opt = match v_opt with
    Some v -> f v
  | None   -> ()
```

Let us move on to the second function, map (not to be confused with List.map, this is Option.map). It takes as arguments a function that takes 'a values and turns them into 'b values, and also a 'a option value. It needs to produce a b option value. So again there's not that many options. If the 'a option value is actually a Some v, then we can apply the function to that v. Otherwise we better return None:

[2] https://github.com/skiadas/TheoryCompCourse

```
let map f v_opt = match v_opt with
    Some v -> Some (f v)
  | None   -> None
```

Moving to the function default. The idea of it is simple: It takes an option value, and as a first argument a "default" value. It delivers that default value if the option value is None, or the value v if the option value is Some v. Here's the code:

```
let default dflt v_opt = match v_opt with
    Some v -> v
  | None   -> dflt
```

The implementations of the other functions are similarly straightforward. Check out the ml file for details. But before we move on, we will introduce one last function:

```
(* val bind : ('a -> 'b option) -> 'a option -> 'b option *)
let bind f v_opt = match v_opt with
    Some v -> f v
  | None   -> None
```

This function takes as input 2 things. The second argument is a value of type 'a option. Think of that as a value that came to us from a previous process, and that process might have failed. The first argument is a function that takes 'a things and processes them, and it also might fail (hence returns 'b option). We essentially combine these two processes, that each have a chance to fail. And if any one of them fails then the overall result is also a failure.

Okay, time to move on.

Let us now compile! Get back out to the terminal, and type:

```
ocamlc -c option.mli option.ml
```

This will create two new "compiled" files, option.cmi and option.cmo. One is the compiled interface file, the other is the compiled object file from option.ml.

Now you can start utop, and from it you can do:

```
#load "option.cmo";;
```

You should now have access to the Option module, notice the capitalization, and its methods, like Option.map, Option.get, Option.default etc.

Custom types

Now that you are getting the hang of modules and stuff, it is time to introduce custom types. The ability to create your own types offers an immense richness to the language and a depth to the type system.

To illustrate custom types, we are going to create an "expression" type, that will represent arithmetic expressions. For example if you think of (a + b) * c, we would have a way to represent this in our type. Here's how such a definition might look like:

```
type t = Num of int
       | Sum of t * t          (* addition *)
       | Prod of t * t         (* multiplication *)
       | Neg of t              (* negation *)
```

Let's read what this says: Something of type t could be one of 4 different things. It could be a Num i where i is of type int, or it could be a Sum (e1, e2) where (e1, e2) is of type t * t, i.e. e1 and e2 are both of type t, or it could be a product etc. If you are thinking of a language like Java, t could be a superclass and each of the other 4 types could be subclasses. OCAML keeps these closer to each other, so to speak.

For instance we can express something like (2 + 4) * 5 as: Prod (Sum (Num 2, Num 4), Num 5).

Note that this type is "recursively defined": Each component refers back to the original type, except for Num of int. You can think of that as the building block, out of which you can create more complicated things by putting existing blocks together.

You might wonder why we called the type t, which sounds very non-descript. We are likely to put the whole thing inside a module, in a file perhaps named expr.ml, so the actual name of the type to the rest of the world would be Expr.t. It is customary to use t for the primary type of a module. It offers some conveniences when trying to share interfaces across multiple modules. For example we might have a "collection" interface, that does not actually implement any functions but declares the kind of function types that collections should have. It is convenient in that case to agree on a common name for the type in question, leaving the module name as the distinguishing factor.

Now let us write our first function that does something with values of type t! Probably the simplest function you could think of is the one that *evaluates* the expression to get a number. So this function will have type:

```
val eval: t -> int
```

Let's think of what it will have to do. It receives as input a value of type t, and that value can be in one of 4 different forms. so it will probably need to do a pattern match against those 4 different forms. Also those forms might contain other expressions of type t, and those would need to be evaluated in turn, by calling our function eval from within itself. So it will have to be recursive:

```
let rec eval e = match e with
   Num i          -> i
 | Sum (e1, e2)   -> eval e1 + eval e2
 | Prod (e1, e2)  -> eval e1 * eval e2
 | Neg e1         -> -(eval e1);;
```

And voila! You just wrote your first interpreter. Here is an example use of it:

```
eval (Prod (Sum (Num 2, Num 4), Num 5));;
```

Let us write another function on values of type t. This function will count how many Num i expressions are present in its argument. Here's how that might look:

```
let rec countNums e = match e with
   Num i          -> 1
 | Sum (e1, e2)
```

```
  | Prod (e1, e2) -> countNums e1 + countNums e2
  | Neg e1        -> countNums e1;;
countNums (Prod (Sum (Num 2, Num 4), Num 5));;
```

Here is something interesting! If we now add a new clause to our type, maybe a variable or something. This would break all these other functions, and we will in fact get a notice about it! When we type these functions in they will complain about inexhaustive matches, i.e. that the different patterns provided do not exhaust the possibilities. This is one of the huge advantages that the type system can provide you, and that you won't get from something like Java's subclasses.